Design of the OS Debugger Daemon

Laurence S. Kaplan
Tera Computer Company
400 N. 34th St.
Seattle, WA 98103

June 2, 1993

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

19970512 068

1 Introduction

In order to facilitate the timely development of the Tera Operating System (OS), the ability to perform source level debugging of that OS is a critical requirement. Additionally, a source level debugger will be of great help in diagnosing future system problems both in-house and at customer sites.

This document describes the portion of the OS debugger that resides on the Tera machine and communicates with a GDB style front end running on some Sun workstation. The front end will often be running on the maintenance workstation although it could be run on any Sun workstation so long as a data path were made available to the Tera machine. The Tera native portion of the debugger is referred to as the GDB tele-debug symbiote, in GDB parlance, or as the Tera OS debugger daemon as it will be refered to in this document.

A principle goal in implementing this daemon is to provide a debugging capability that is operable even when most of the OS might not be. To accomplish this, the daemon must be as self sufficient as possible, at least to the extent that it need not rely on any OS services. It may use those services, but should have various fallbacks available to provide some minimum functionality.

2 Debugger Software Architecture

The Tera debugger (tdb), in its early implementation, exists as a process running on a Sun Workstation. Currently, the debugger connects directly to a tele-debug symbiote that is part of the Zebra simulator via a pseudo-tty (eventually a socket). This connection provides a transport mechanism for requests from the debugger to Zebra for various kinds of information. These requests can include things like "read register r5 from HW stream 3 on processor 1" or "read memory location 0x10000 as seen by HW stream 3 on processor 1". Such registers and memory can also be written via these requests.

In order to prepare for the move to real HW, the Tera OS debugger daemon will respond to the same set of queries as the tele-debug symbiote within Zebra, without requiring access to any internal Zebra data. By using the same transport mechanisms and protocols, most of the code can be shared by the two modes of debugging. The debugger daemon can be used with Zebra, along with

DTIC QUALITY INSPECTED 3

or instead of the Zebra gdb symbiote. However, on the real hardware, only the debugger daemon method is available for use with tdb.

2.1 Daemon Architecture

Various requests made of the debugger daemon will require that it be running on the correct processor. For this reason it is likely that there will be a copy of the debugger daemon running on every processor at least at some points during system execution. Some coordination of these daemons will be necessary. One will likely act as the main daemon responsible for communication with tdb and the others will communicate through the main one.

As mentioned earlier, it is essential that the debugger daemon be able to operate on a fairly corrupted operating system. In order to protect the daemon (and the OS microkernel), all data mappings of daemon (and microkernel) text will be read-only. If breakpoints need to be placed using this data mapping, it will be given write permision for the smallest possible duration.

2.2 User Mode Support

Due to the nature of the Tera OS and user mode runtime system, there will be many complicated interactions between the OS and user mode programs. For this reason, it is necessary that the debugger daemon and tdb be able to debug both the OS and an arbitrary user mode program at the same time. This creates a couple of problems. The debugger daemon needs to properly synchronize with the user mode program, most likely by communicating with the OS scheduler. Tdb needs to be able to handle more than one symbol table at a time, where these symbol tables are guaranteed to be disjoint with respect to virtual addresses, but probably not disjoint with respect to symbol names. It is likely that this will be more easily implemented by using two tdb front-ends multiplexing to the same debugger daemon.

3 Basic Debugging Requests

What follows is a discussion of the implementation of the various basic requests that tdb might make of the debugger daemon. While the current Zebra symbiote can obtain or modify most of this information directly using Zebra internal state, the OS debugger daemon must use other mechanisms. Note that a lot of other functionality relating to OS debugging is desirable in the front-end. Those features are not discussed here.

3.1 Identifying the Entity to be Debugged

Currently, tdb handles multiple streams by specifying a processor and hardware stream number to uniquely identify a stream. This is necessary since some types of user mode streams are anonymous, i.e. there are no SW identifiers generated for these streams. In the OS, this is not the case. Every stream within the OS has an associated unique `OS_chore_ID`. In addition, it is not possible for the debugger daemon to use HW stream numbers since this does not help in identifying streams within the OS. Basically, when running on the HW, there is no way to say "show me HW stream 5" though Zebra could handle such a request. The current proposal is to use the OS chore IDs to identify OS streams. Given a chore ID, the debugger daemon is able to look up the `OS_Task` containing the stream since the first part of a chore ID is a task index. This task data structure contains a set of team chore lists of all the privileged chores in the task. In order to actually look at the chore control blocks (CCBs) in the the chore list, the daemon must enter the domain where the chore's team is loaded since CCBs are allocated in privileged private memory. The daemon can then check each CCB in each team list for the desired chore id.

3.2 Reading and Writing the Address Space

Once the stream/chore has been identified, reading and writing the address space is accomplished by entering the correct domain on the correct processor for the stream and using the address maps that are there. For privileged common memory, domain movement is not needed, though the correct processor is often still needed. As stated before, privileged private memory references for things like CCBs do require domain movement.

One slightly tricky aspect of this functionality is dealing with state bits. Memory reads (and writes) will often care about state bits. The code will at least need to know how to deal with the "locked" state. It will also need to be able to read and write state bits on demand. It may be that all memory transaction will be performed using the STATE load and store instructions. For any transactions requiring state bits, the state will need to be locked prior to reading or writing the state and value of the location, and then unlocked afterwards. This is because two instructions are needed to read the memory, one to read the state, one to read the value, and these two separate instructions must be made to appear atomic.

This is a prime area for the debugger daemon to have a fallback strategy in case the domain's regular HW address maps are corrupted. In this case the debugger daemon can construct its own mapping based on the SW data structures for the address space. Significant work will be needed in the debugger daemon's data exception handlers to make this kind of request fully robust. It would also be helpful to have routines to check map consistency. This could include checking the HW maps against the SW maps. Such a verification might be performed automatically after a panic prior to accepting requests from the tdb front-end.

3.3 Reading and Writing the Processor Registers

Since OS chores, especially kernel level chores, are not arbitrarily preemptable, it is not possible to simply stop the stream in question to examine its register state. The only chores for which this is possible are those that are already blocked of their own volition. For all other chores, the domain signal can be used to cause the stream to dump its registers into its chore control block (CCB) and then continue on where it left off. Additionally, the traps used to implement breakpoints and watchpoints (described below) will also dump stream register state to the stream's CCB. To write registers, a special command will be left in the CCB that will cause the trap handler to modify the register. Note that while domain signal will not normally be used to write registers, the trap handlers used for breakpoints and watchpoints will often be used for this. As with memory operations, register operations also need to deal with the register tag bit "poison".

Care must be taken when using domain signal for various reasons. First, the debugger daemon itself should be immune to the signal. This is probably easily done by simply masking the trap. Next, if the domain signal is asserted in any domain other than the OS domain, any user mode streams running in the domain will think that they just got a PM-swap request or UNIX signal. Some synchronization may be needed with the scheduler to somehow suspend the user mode streams so that they do not interfere with the debugging of promoted streams and visa versa. Also, the domain signal can only be asserted from the processor containing the domain.

3.4 Breakpoints

Breakpoints are likely to be one of the most complicated features of the OS debugger. Breakpoints are normally set by writing some form of trap or illegal instruction into the text space of the task. In serial machines, when a breakpoint is hit, the instruction that was replaced by the trap is executed by replacing the instruction back where it belongs, single stepping the process, and then re-inserting the breakpoint trap. Unfortunately, this is not appropriate in a multiprocessing environment since other streams might fly by the breakpoint while the original instruction is temporarily there. So a more complicated method will be required.

The current leading proposal is to leave the invalid instruction in place and put the real instruction somewhere else in memory. The instruction counter can then be used to single step the one instruction wherever it may be. The only problem with this is for program counter (PC) relative operations. These include skip and target_disp operations. If these were the only PC relative operations, they could be handled by examining the target registers and PC after the instruction executes in order to see if they picked up any values based on the new virtual address for the replaced instruction. Unfortunately, there is also an ssw_disp instruction that can put a PC value into a general purpose register. Since it would be impossible to distinguish between random data and a PC value in a general purpose register, simply examining registers after the instruction executes is insufficient. This currently leaves full emulation of PC relative operations as the only solution. This facility needs more design work for both the OS and real user mode debugging.

Additional code will need to be added to the illegal instruction handler in order to coordinate with the debugger and to restart those streams that are not of interest. Note that breakpointing streams that are not supposed to block in a multi-stream task is dangerous in that other streams may be trying to synchronize with those at breakpoints. Additionally, in the OS, various scheduler and other team related synchronizations may suffer if a member stream is sitting at a breakpoint. This handler will also have code in it similar to the domain signal handler for dumping and restoring register state. This allows registers to be read and written while sitting at breakpoints.

Native support for conditional breakpoints is also very desirable. Instead of requiring that the front-end evaluate the condition everytime a breakpoint is reached, it would be much more efficient if the debugger daemon or trap handler could do the condition evaluation.

3.5 Continuing and Stepping

The single stepping of OS streams can probably be handled in the normal fashion of using the built-in instruction counter. Continuing should also be relatively easy once the problems of stopping a stream have been addressed. Whichever trap handler deals with stopping the stream will also likely be responsible for restarting it.

3.6 Unexpected Exceptions

The trap handlers for OS streams should also know how to notify the debugger daemon of unexpected events. This would allow things like memory addressing errors to be reported in a timely fashion to the front end debugger. The daemon should also be able to handle its own random failures whenever it is trying to use microkernel services or data structures. Ideally, there should always be a fallback for every operation. Realistically, there will be some operations whose failures will be uncorrectable.

Since the debugger daemon will be replicated to run on multiple processors, there will be more than one copy of its text segment. It may be possible to reload a debugger daemon on one processor from another if the first becomes corrupted. Much care will be needed to properly detect and handle this.

4 Tera Extensions

The Tera architecture suggests some additional functionality in a debugger. The following features will be used by the Tera user level debugger and should also be available for the OS debugger.

4.1 Watchpoints

The Tera memory architecture provides two data trap bits that cause a stream to raise an exception if one of these bits is observed to be set during a memory transaction. By convention, one of these bits is reserved for use to implement continuation points. Watchpoints are a special case of continuation points where the routine that gets continued will notify the debugger daemon of the reference. A watchpoint allows a user to ask the debugger to find out what stream next references a particular memory location. This support will also be available in the OS. Some work will be needed in the OS data exception handlers to support this. Note that this exception handler also needs the register save and restore code that the domain signal and instruction exception handlers contain.

4.2 IOP Debugging

Some provision needs to be made for debugging the IOPs. These processors have very little internal state. The IOP drivers that run on the regular processors will probably know most of the internal state of the IOPs they control. For more serious problems, the scan system will be used. It is important that the debugger knows how to disassemble IOP programs and communicate with the IOP drivers to obtain IOP state. It would also be useful to allow the debugger to ask for an IOP reset.

5 Debugging Environments

There are two environments under which the operating system can be debugged using tdb and the debugger daemon. One is the Zebra simulator running on some simulation engine (currently a Sun). This environment has the benefit that tdb can also communicate directly with Zebra while the debugger daemon is being developed. The other environment is the Tera machine itself. Here, the only fallback if the debugger daemon fails is the scan subsystem.

5.1 Simulator

Hopefully, most of the code written to support OS debugging on the simulator will be directly useful on the real machine. The only additional code necessary is that to interface the debuggers with Zebra itself. This is needed for debugging user mode code on Zebra anyway.

5.2 Tera Hardware

This is the hardest and most important platform for debugging the OS. The current plan is to provide a serial interface through a HIPPI channel that can communicate with the main debugger daemon running on one of the Tera processors. The daemon will probably run at the kernel protection level in order to be as immune as possible to problems in the system. This also allows it to avoid having any privilege problems. To keep it from being affected by the domain signal, it will probably leave that trap masked. An important point to reiterate is that the debugger daemon should depend as little as possible on the rest of the OS. This includes not always relying on the normal system address maps. The daemon probably will need its own dedicated segments to use to poke at various parts of memory when the regular maps are in a questionable state. The daemon will, however, likely be part of the OS image.

The front-end debugger, in this situation, can probably run on any machine that is networked with the maintenance machine, which will be a Sun. By only using the maintenance machine as a debugger communications server, a partitioned machine can have multiple partitions being debugged at once. It may also be desirable to allow the debugger to be running native on the Tera being debugged. Though this is not so useful for crashed or ailing systems, it can be very useful for performance analysis and other behavior investigations.

6 Special Operating System Situations

There are a few special situations that can occur in the OS that deserve special mention with relation to debugging. Two are system crashes and hangs. Another is system call debugging with user mode interactions. This section will evolve as we come to understand more of the Tera system characteristics.

6.1 System Call Debugging

As mentioned earlier, there will be many situations when a developer will want to debug the OS and some user mode program together, in order to debug some tricky OS/runtime interactions. In this scenario, the developer might set breakpoints in the user mode program at the system call invocation and some of the runtime upcall protocol routines. Breakpoints would also be set within the OS in the system call being used and in the upcall implementation in the OS. The developer would then be able to track the progress of the system call through any chore blocks and unblocks. This type of debugging session will also be applicable for signal handling and swap debugging.

6.2 System Crashes

When the OS crashes or panics, it is likely that the debugger daemon will still be operable, so no change in communication is needed. However, it is important to get the entire system quiescent as soon as possible after a crash so that the debugger has some chance of figuring out what went wrong. Some way will be needed to stop all of the processors and enter debugger control. This should include dumping all of the executing streams' register state and ensuring that debugger daemons exist or are runnable on all processors. This would also be a good time for the debugger daemon to run some consistency checks on itself and other important data structures to ensure that nothing has been corrupted. Even if corruption is detected, it should be handled gracefully so that some amount of debugging can still be performed.

6.3 Partial System Hangs

When part of the OS hangs, e.g. there is some form of deadlock or lack of progress in parts of the OS, enough of the system should still be active to proceed with normal debugging. Whether or not to stop other activity on the system is unclear.

6.3.1 Kernel DeadLock

One of the possible causes of a partial system hang might be deadlock on some spin lock or spin event between two kernel streams. In order to diagnose this problem a developer would probably start by querying the system about where all of the stream are using traceback information. This would be obtained by using the domain signal to get the current state of the streams and then using that information to backtrace each stream. From there the developer should be able to identify the locks or event involved and then look at those data structures to find the deadlock.

6.4 Total System Hangs

In the rare situation when the OS hangs completely, there is a chance that the debugger daemon will not be operational. In this case we fall back to the scan system to try and find the reason for the hang. The scan facility is beyond the scope of this document. It should be possible, however, to use scan to restart a debugger daemon. How well connected to the OS's data structures such a restarted daemon will be remains to be seen.